A Type Soundness Proof for Variables in LCF ML [†]

Dennis Volpano
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943, USA
volpano@cs.nps.navy.mil

Geoffrey Smith
School of Computer Science
Florida International University
Miami, FL 33199, USA
smithg@fiu.edu

Abstract

We prove the soundness of a polymorphic type system for a language with variables, assignments, and first-class functions. As a corollary, this proves the soundness of the Edinburgh LCF ML rules for typing variables and assignments, thereby settling a long-standing open problem.

Keywords: Type theory, formal semantics, variables and assignment.

1 Introduction

A type system is presented for a language with a `letvar` construct to allocate *variables*, which are implicitly dereferenced and whose addresses are not first-class values, as in traditional imperative languages. Edinburgh LCF ML [GMW78] had such a construct, which it called `letref`. We show that the restriction that a variable must have weak type only if it is assigned to inside a λ-abstraction within its scope is sound. As a corollary then, LCF ML restriction 2(i)(b) (pg. 49 [GMW78]), which requires a variable to have a monotype (a type with no type variables) if the variable is assigned to inside a λ-abstraction within its scope, is also sound since every monotype is weak. This restriction was never proved sound, according to Tofte [Tof90].

2 The Type System

The syntax of the language we consider is core ML with a **letvar** construct and assignment. Following Tofte [Tof90], we distinguish a subset of the expressions called *Values*:

[†] Appeared in Information Processing Letters, 56(3), November 1995, pp.141–146. This material is based upon activities supported by the National Science Foundation under Agreements No. CCR-9400592 and CCR-9414421. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **01 NOV 1995**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**

4. TITLE AND SUBTITLE **A Type Soundness Proof for Variables in LCF ML**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Department of Computer Science Naval Postgraduate School Monterey, CA 93943**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**9**	

$$
\begin{array}{lrcl}
(\mathit{Expressions}) & e & ::= & v \mid l \mid e_1\, e_2 \mid e_1 := e_2 \mid \\
 & & & \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid \\
 & & & \mathbf{letvar}\ x := e_1\ \mathbf{in}\ e_2 \\
(\mathit{Values}) & v & ::= & x \mid \mathbf{unit} \mid \lambda x.\, e
\end{array}
$$

Meta-variable x ranges over identifiers. The **letvar** construct binds x to a new cell initialized to the value of e_1. The scope of the binding is e_2 and the lifetime of the cell is unbounded. Dereferencing of variables created with **letvar** is implicit. Locations are denoted by meta-variable l and are not values.

The types of the language are stratified as follows.

$$
\begin{array}{rcll}
\tau & ::= & \alpha \mid \mathit{unit} \mid \tau \rightarrow \tau' & (\mathit{data\ types}) \\
\sigma & ::= & \forall \alpha \,.\, \sigma \mid \tau & (\mathit{type\ schemes}) \\
\rho & ::= & \sigma \mid \tau\ \mathit{var} & (\mathit{phrase\ types})
\end{array}
$$

The meta-variable α ranges over *type variables.* Type variables are partitioned into *weak* and *strong* type variables, written $\_\alpha$ and α respectively. These variables correspond to the imperative and applicative type variables respectively of Tofte's system. We say that a type scheme σ is *weak* iff σ is unquantified and every type variable in σ is weak. Type $\tau\ \mathit{var}$ is the type of locations storing values of type τ.

The rules of the type system are formulated as they are in Harper's system [Har94] and are given in Figure 1. It is a deductive proof system used to assign types to expressions. Typing judgements have the form

$$\lambda; \gamma \vdash e : \rho$$

meaning that expression e has type ρ assuming that λ prescribes type schemes for locations in e and γ prescribes phrase types for the free identifiers of e. Meta-variable γ ranges over identifier typings. An *identifier typing* γ is a finite function mapping identifiers to phrase types; $\gamma(x)$ is the phrase type assigned to x by γ and $\gamma[x : \rho]$ assigns phrase type ρ to x and to variable $x' \neq x$, phrase type $\gamma(x')$.

Meta-variable λ ranges over location typings. Unlike other approaches [Tof90, Har94, SmVo95], a *location typing* here is a finite function mapping locations to *type schemes.* This is the most novel aspect of the type system. The notational conventions for location typings are similar to those for identifier typings.

The *generalization* of a type scheme σ relative to λ and γ, written $\mathit{Close}_{\lambda;\gamma}(\sigma)$, is the type scheme $\forall \bar{\alpha} \,.\, \sigma$, where $\bar{\alpha}$ is the set of all type variables occurring free in σ but not in λ or in γ. We write $\lambda \vdash e : \tau$ and $\mathit{Close}_{\lambda}(\sigma)$ when $\gamma = \emptyset$. A restricted form of generalization, written $\mathit{AppClose}_{\lambda;\gamma}(\sigma)$, is defined to be the same as $\mathit{Close}_{\lambda;\gamma}(\sigma)$ except that only strong type variables are generalized; any weak ones remain free. As in Tofte [Tof90], the *generic instance* relation ($\geq$) of Damas and Milner [DaM82] is restricted by requiring universally quantified weak type variables to be instantiated only with weak types.

Finally, we write $\lambda; \gamma \vdash e : \sigma$ iff $\lambda; \gamma \vdash e : \tau$ whenever $\sigma \geq \tau$.

3 Semantics and Soundness

In this section, we establish type soundness using the framework of Harper [Har94]. First we give a structured operational semantics for the language. An expression is evaluated

(VAR) $\lambda;\gamma \vdash x : \tau\ \mathit{var} \qquad \gamma(x) = \tau\ \mathit{var}$

(IDENT) $\lambda;\gamma \vdash x : \tau \qquad \gamma(x) \geq \tau$

(VARLOC) $\lambda;\gamma \vdash l : \tau\ \mathit{var} \qquad \lambda(l) \geq \tau$

(UNIT) $\lambda;\gamma \vdash \mathbf{unit} : \mathit{unit}$

($\rightarrow$-INTRO) $$\frac{\lambda;\gamma[x : \tau_1] \vdash e : \tau_2}{\lambda;\gamma \vdash \lambda x.\, e : \tau_1 \rightarrow \tau_2}$$

($\rightarrow$-ELIM) $$\frac{\lambda;\gamma \vdash e_1 : \tau_1 \rightarrow \tau_2,\ \ \lambda;\gamma \vdash e_2 : \tau_1}{\lambda;\gamma \vdash e_1\ e_2 : \tau_2}$$

(LET-VAL) $$\frac{\lambda;\gamma \vdash v : \tau_1,\ \ \lambda;\gamma[x : \mathit{Close}_{\lambda;\gamma}(\tau_1)] \vdash e : \tau_2}{\lambda;\gamma \vdash \mathbf{let}\ x = v\ \mathbf{in}\ e : \tau_2}$$

(LET-ORD) $$\frac{\lambda;\gamma \vdash e_1 : \tau_1,\ \ \lambda;\gamma[x : \mathit{AppClose}_{\lambda;\gamma}(\tau_1)] \vdash e_2 : \tau_2}{\lambda;\gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2}$$

(LETVAR) $$\frac{\begin{array}{l}\lambda;\gamma \vdash e_1 : \tau_1,\ \ \lambda;\gamma[x : \tau_1\ \mathit{var}] \vdash e_2 : \tau_2 \\ \text{If } x \text{ is assigned to in a } \lambda\text{-abstraction in } e_2 \text{ then } \tau_1 \text{ is weak.}\end{array}}{\lambda;\gamma \vdash \mathbf{letvar}\ x := e_1\ \mathbf{in}\ e_2 : \tau_2}$$

(R-VAL) $$\frac{\lambda;\gamma \vdash e : \tau\ \mathit{var}}{\lambda;\gamma \vdash e : \tau}$$

(ASSIGN) $$\frac{\lambda;\gamma \vdash e_1 : \tau\ \mathit{var},\ \ \lambda;\gamma \vdash e_2 : \tau}{\lambda;\gamma \vdash e_1 := e_2 : \mathit{unit}}$$

Figure 1: Rules of the Type System

$$(\text{VAL}) \qquad \mu \vdash v \Rightarrow v, \mu$$

$$(\text{DEREF}) \qquad \mu \vdash l \Rightarrow \mu(l), \mu$$

$$(\text{APPLY}) \qquad \frac{\begin{array}{l} \mu \vdash e_1 \Rightarrow \lambda x.\, e_1{}', \mu_1 \\ \mu_1 \vdash e_2 \Rightarrow v_2, \mu_2 \\ \mu_2 \vdash [v_2/x]e_1{}' \Rightarrow v, \mu' \end{array}}{\mu \vdash e_1\ e_2 \Rightarrow v, \mu'}$$

$$(\text{UPDATE}) \qquad \frac{\mu \vdash e \Rightarrow v, \mu'}{\mu \vdash l := e \Rightarrow \mathbf{unit}, \mu'[l := v]}$$

$$(\text{BIND}) \qquad \frac{\begin{array}{l} \mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ \mu_1 \vdash [v_1/x]e_2 \Rightarrow v_2, \mu_2 \end{array}}{\mu \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Rightarrow v_2, \mu_2}$$

$$(\text{BINDVAR}) \qquad \frac{\begin{array}{l} \mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ l \notin dom(\mu_1) \\ \mu_1[l := v_1] \vdash [l/x]e_2 \Rightarrow v_2, \mu_2 \end{array}}{\mu \vdash \mathbf{letvar}\ x := e_1\ \mathbf{in}\ e_2 \Rightarrow v_2, \mu_2}$$

Figure 2: The Evaluation Rules

relative to a *memory* μ, which is a finite function from locations to values. The contents of a location $l \in dom(\mu)$ is the value $\mu(l)$, and we write $\mu[l := v]$ for the memory that assigns value v to location l, and value $\mu(l')$ to a location $l' \neq l$. Note that $\mu[l := v]$ is an *update* of μ if $l \in dom(\mu)$ and an *extension* of μ if $l \notin dom(\mu)$. The *range* of μ is the set of all values $\mu(l)$, for $l \in dom(\mu)$.

Our evaluation rules are given in Figure 2. They allow us to derive judgements of the form

$$\mu \vdash e \Rightarrow v, \mu'$$

which is intended to assert that evaluating closed expression e in memory μ results in value v and new memory μ'. We write $[e'/x]e$ to denote the capture-avoiding substitution of e' for all free occurrences of x in e. The use of substitution in the rules allows us to avoid environments and closures in the semantics, so that the result of evaluating an expression is just another expression.

The basic idea behind showing soundness is to show that if $\vdash e : \tau$ and $\vdash e \Rightarrow v, \mu'$, then $\vdash v : \tau$, a property called *subject reduction*. But since e can allocate locations and since these locations can occur in v, the conclusion must actually be that there exists a location typing λ' such that $\lambda' \vdash v : \tau$ and such that $\mu' : \lambda'$. The latter condition asserts that λ' is consistent with μ'. More precisely we say that $\mu : \lambda$ iff $dom(\mu) = dom(\lambda)$ and for every $l \in dom(\mu)$, $\lambda \vdash \mu(l) : \lambda(l)$.

It is the location typing λ' that makes soundness delicate. As observed by Tofte [Tof90], we may generalize a type variable α in typing $\vdash e : \tau$, only to find that α occurs free in λ', and therefore cannot be generalized in typing $\lambda' \vdash v : \tau$. For example, we can define list reversal as follows:

```
letvar $r := \lambda x.\, x$ in
    $r := \lambda x.$ if $x = [\,]$ then $[\,]$ else $(r\ (tl\ x))$ @ $[hd\ x]$;
    $r$
end
```

This expression has type $\forall\alpha\,.\,\alpha\ list \rightarrow \alpha\ list$ in our type system. But when the expression is evaluated, a location l of type $\alpha\ list \rightarrow \alpha\ list$ is allocated for r and l appears in the resulting value as well as in the domain of the resulting location typing λ'.

The solution proposed here is to use the *quantified* type $\forall\alpha\,.\,\alpha\ list \rightarrow \alpha\ list$ for l in λ', thereby eliminating the free occurrence of α. Of course, it is not always reasonable to give a location a quantified type. For example, if $\lambda(l) = \forall\alpha\,.\,\alpha \rightarrow \alpha$, then the program $l := not;\ l$ can be given type $int \rightarrow int$, yet it evaluates to not of type $bool \rightarrow bool$. Our subject reduction theorem allows only *read-only* locations to be given quantified types.

We now turn to the soundness proof. First we introduce the relevant lemmas.

Lemma 3.1 (Superfluousness) *Suppose that* $\lambda;\gamma \vdash e : \tau$. *If* $l \notin dom(\lambda)$, *then* $\lambda[l : \sigma];\gamma \vdash e : \tau$ *and if* $x \notin dom(\gamma)$, *then* $\lambda;\gamma[x : \sigma] \vdash e : \tau$.

Lemma 3.2 (Substitution) *If* $\lambda;\gamma \vdash v : \sigma$ *and* $\lambda;\gamma[x : \sigma] \vdash e : \tau$, *then* $\lambda;\gamma \vdash [v/x]e : \tau$. *Also, if* $\lambda;\gamma \vdash l : \tau\ var$ *and* $\lambda;\gamma[x : \tau\ var] \vdash e : \tau'$, *then* $\lambda;\gamma \vdash [l/x]e : \tau'$.

The preceding two lemmas are straightforward variants of the lemmas given in [Har94]. We also need two new lemmas:

Lemma 3.3 (Strengthening) *If* $\lambda[l : \sigma_1] \vdash e : \sigma$ *and* $\sigma_2 \geq \sigma_1$ *then* $\lambda[l : \sigma_2] \vdash e : \sigma$.

Lemma 3.4 ($\forall$-intro) *If* $\lambda \vdash e : \sigma$ *and* α *does not occur free in* λ, *then* $\lambda \vdash e : \forall\alpha\,.\,\sigma$.

Finally, we note that in spite of (R-VAL) our typing rules are essentially "syntax directed":

Lemma 3.5 ((R-VAL)-scope) *If the derivation of* $\lambda;\gamma \vdash e : \tau$ *ends with* (R-VAL), *then* e *is an identifier or a location.*

Proof. If the derivation ends with (R-VAL), then there must be a derivation of the hypothesis $\lambda;\gamma \vdash e : \tau\ var$. But to show that an expression has a type of the form $\tau\ var$, there are only two possible rules that can be used: (VAR) and (VARLOC). (The other rules all give *data types* to expressions.) So e must either be an identifier, in the case of (VAR), or a location, in the case of (VARLOC). □

We now give the soundness theorem:

Theorem 3.6 (Subject Reduction) *Suppose*

(*a*) $\mu \vdash e \Rightarrow v, \mu'$,
(*b*) $\lambda \vdash e : \tau$,
(*c*) $\mu : \lambda$, *and*
(*d*) *if a location* l *is assigned to in* e, *then* $\lambda(l)$ *is unquantified; also, if* l *is assigned to in the range of* μ *or in a* λ*-abstraction in* e, *then* $\lambda(l)$ *is weak.*

Then there exists λ' such that

(e) $\lambda \subseteq \lambda'$,
(f) $\mu' : \lambda'$,
(g) $\lambda' \vdash v : \tau$,
(h) *any strong type variable free in λ' is free in λ, and*
(i) *if a location l is assigned to in v or in the range of μ', then $\lambda'(l)$ is weak.*

Proof. The proof is by induction on the structure of the derivation of $\mu \vdash e \Rightarrow v, \mu'$. Due to space limitations, we present only the most interesting cases, (UPDATE) and (BINDVAR). We remark that property (h) above makes the (BIND) case routine.

(UPDATE). The evaluation must end with

$$\frac{\mu \vdash e \Rightarrow v, \mu'}{\mu \vdash l := e \Rightarrow \mathbf{unit}, \mu'[l := v]}$$

and, by Lemma 3.5, the typing must end with

$$\frac{\lambda \vdash l : \tau\ var, \quad \lambda \vdash e : \tau}{\lambda \vdash l := e : unit}$$

Also, $\mu : \lambda$, $\lambda(l)$ is unquantified, and if a location l' is assigned to in e, then $\lambda(l')$ is unquantified. And if l' is assigned to in the range of μ or in a λ-abstraction in e, then $\lambda(l')$ is weak. By induction, there exists λ' such that

(e) $\lambda \subseteq \lambda'$,
(f) $\mu' : \lambda'$,
(g) $\lambda' \vdash v : \tau$,
(h) any strong type variable free in λ' is free in λ, and
(i) if a location l' is assigned to in v or in the range of μ', then $\lambda'(l')$ is weak.

Now we must show

(f) $\mu'[l := v] : \lambda'$,
(g) $\lambda' \vdash \mathbf{unit} : unit$,
(i) if a location l' is assigned to in **unit** or in the range of $\mu'[l := v]$,
then $\lambda'(l')$ is weak.

(g) follows immediately from typing rule (UNIT). (i) follows by induction, since if a location l' is assigned to in the range of $\mu'[l := v]$ then it is assigned to in v or in the range of μ'. Finally, we consider (f), the most interesting case. For every $l' \in dom(\mu')$ and $l' \neq l$, we have

$$\lambda' \vdash \mu'[l := v](l') : \lambda'(l')$$

by induction. Since $\lambda \vdash l : \tau\ var$, $\lambda(l) \geq \tau$. But since $\lambda(l)$ is *unquantified*, $\lambda(l) = \tau$ and therefore $\lambda'(l) = \tau$ since $\lambda \subseteq \lambda'$. Since, by induction, $\lambda' \vdash v : \tau$, we have

$$\lambda' \vdash \mu'[l := v](l) : \lambda'(l)$$

Thus we have $\mu'[l := v] : \lambda'$. This completes (UPDATE).

Notice the role of condition (d) in proving $\lambda' \vdash \mu'[l := v](l) : \lambda'(l)$ above. Since l is assigned to in $l := e$, $\lambda(l)$ must be unquantified and consequently has only one generic instance, namely τ. Therefore, $\lambda' \vdash \mu'[l := v](l) : \lambda'(l)$ follows directly from $\lambda' \vdash v : \tau$ of the induction. If $\lambda(l)$ were quantified, then it would not be possible to show $\lambda' \vdash v : \lambda'(l)$. For example, if $\lambda(l) = \forall\alpha \,.\, \alpha \rightarrow \alpha$, then on the program $l := \mathit{not}$ we would have to show that not has type $\forall\alpha \,.\, \alpha \rightarrow \alpha$.

(BINDVAR). The evaluation must end with

$$\frac{\begin{array}{l}\mu \vdash e_1 \Rightarrow v_1, \mu_1 \\ l \notin \mathit{dom}(\mu_1) \\ \mu_1[l := v_1] \vdash [l/x]e_2 \Rightarrow v_2, \mu_2\end{array}}{\mu \vdash \textbf{letvar}\ x := e_1\ \textbf{in}\ e_2 \Rightarrow v_2, \mu_2}$$

and, by Lemma 3.5, the typing must end with

$$\frac{\begin{array}{l}\lambda \vdash e_1 : \tau_1 \\ \lambda; [x : \tau_1\ \mathit{var}] \vdash e_2 : \tau_2 \\ \text{If } x \text{ is assigned to in a } \lambda\text{-abstraction in } e_2 \text{ then } \tau_1 \text{ is weak.}\end{array}}{\lambda \vdash \textbf{letvar}\ x := e_1\ \textbf{in}\ e_2 : \tau_2}$$

Also, $\mu : \lambda$ and if a location l' is assigned to in e_1 or in e_2, then $\lambda(l')$ is unquantified. And if l' is assigned to in the range of μ or in a λ-abstraction in e_1 or in e_2, then $\lambda(l')$ is weak. By induction, there exists λ_1 such that

(e) $\lambda \subseteq \lambda_1$,
(f) $\mu_1 : \lambda_1$,
(g) $\lambda_1 \vdash v_1 : \tau_1$,
(h) any strong type variable free in λ_1 is free in λ, and
(i) if a location l' is assigned to in v_1 or in the range of μ_1, then $\lambda_1(l')$ is weak.

Since $l \notin \mathit{dom}(\lambda_1)$, $\lambda_1 \subseteq \lambda_1[l : \tau_1]$. Now, since $\lambda_1[l : \tau_1] \vdash l : \tau_1\ \mathit{var}$ and, by Lemma 3.1, $\lambda_1[l : \tau_1]; [x : \tau_1\ \mathit{var}] \vdash e_2 : \tau_2$, we can apply Lemma 3.2 to get

(b) $\lambda_1[l : \tau_1] \vdash [l/x]e_2 : \tau_2$

We also have, by Lemma 3.1,

(c) $\mu_1[l := v_1] : \lambda_1[l : \tau_1]$

Next, if a location l' is assigned to in $[l/x]e_2$, then either l' is assigned to in e_2 or $l' = l$. In the first case we have that $\lambda(l')$ is unquantified by hypothesis, and so $\lambda_1[l : \tau_1](l')$ is unquantified. In the second case we have $\lambda_1[l : \tau_1](l) = \tau_1$, which is unquantified. Also, if l' is assigned to in the range of $\mu_1[l := v_1]$, then l' is assigned to in v_1 or in the range of μ_1, so by induction $\lambda_1(l')$ is weak, and hence $\lambda_1[l : \tau_1](l')$ is weak, since $\lambda_1 \subseteq \lambda_1[l : \tau_1]$. Finally, if l' is assigned to in a λ-abstraction in $[l/x]e_2$, then either l' is assigned to in a λ-abstraction in e_2 or $l' = l$ and x is assigned to in a λ-abstraction in e_2. In the first case, $\lambda(l')$ is weak by hypothesis, and so $\lambda_1[l : \tau_1](l')$ is weak. In the second case, we have τ_1 is weak by the restriction on the (LETVAR) rule, and so $\lambda_1[l : \tau_1](l')$ is weak. Therefore, we have

(d) if a location l' is assigned to in $[l/x]e_2$, then $\lambda_1[l:\tau_1](l')$ is unquantified; also, if l' is assigned to in the range of $\mu_1[l := v_1]$ or in a λ-abstraction in $[l/x]e_2$, then $\lambda_1[l:\tau_1](l')$ is weak.

So by a second use of induction, there exists λ_2 such that

(e) $\lambda_1[l:\tau_1] \subseteq \lambda_2$,
(f) $\mu_2 : \lambda_2$,
(g) $\lambda_2 \vdash v_2 : \tau_2$,
(h) any strong type variable free in λ_2 is free in $\lambda_1[l:\tau_1]$, and
(i) if a location l' is assigned to in v_2 or in the range of μ_2, then $\lambda_2(l')$ is weak.

At this point, λ_2 may contain free strong type variables that are not free in λ, namely those of τ_1. So we cannot take λ_2 as our final location typing. Instead, define λ' by

$$\lambda'(l') = \mathit{AppClose}_\lambda(\lambda_2(l')),$$

for all $l' \in \mathit{dom}(\lambda_2)$. Now we must establish

(e) $\lambda \subseteq \lambda'$,
(f) $\mu_2 : \lambda'$,
(g) $\lambda' \vdash v_2 : \tau_2$,
(h) any strong type variable free in λ' is free in λ, and
(i) if a location l' is assigned to in v_2 or in the range of μ_2, then $\lambda'(l')$ is weak.

To show (e), note that for any $l' \in \mathit{dom}(\lambda)$, $\lambda'(l') = \lambda_2(l')$, by the definition of λ'. Since $\lambda \subseteq \lambda_2$, it follows that $\lambda \subseteq \lambda'$.

Next we show (f). Since $\mu_2 : \lambda_2$, we have

$$\lambda_2 \vdash \mu_2(l') : \lambda_2(l')$$

for any $l' \in \mathit{dom}(\mu_2)$. Since $\mathit{AppClose}_\lambda(\sigma) \geq \sigma$ for every σ, by applying Lemma 3.3 repeatedly we get

$$\lambda' \vdash \mu_2(l') : \lambda_2(l')$$

Finally, from Lemma 3.4 we get

$$\lambda' \vdash \mu_2(l') : \mathit{AppClose}_\lambda(\lambda_2(l')),$$

since any type variables thereby quantified do not occur free in λ'. Hence $\mu_2 : \lambda'$.

To get (g), apply Lemma 3.3 to $\lambda_2 \vdash v_2 : \tau_2$. And ($h$) follows immediately from the definition of λ'. Finally, for (i) suppose that l' is assigned to in v_2 or in the range of μ_2. By the second use of induction, $\lambda_2(l')$ is weak. Hence $\lambda'(l')$ is weak, since *AppClose* quantifies only strong type variables. This completes (BINDVAR). □

Corollary 3.7 *Restriction 2(i)(b) of LCF ML [GMW78], requiring a variable to have a monotype if the variable is assigned to in a λ-abstraction within its scope, is sound.*

Proof. A monotype is a type with no type variables, so every such type is weak. So by Theorem 3.6, the LCF ML restriction is sound. □

References

[DaM82] Damas, L. and Milner, R., Principal Type Schemes for Functional Programs, *Proc. 9th ACM Symposium on Principles of Programming Languages*, pp. 207–212, 1982.

[GMW78] Gordon, M., Milner, A. and Wadsworth, C., Edinburgh LCF, *Lecture Notes in Computer Science* **78**, Springer-Verlag, 1979.

[Har94] Harper, R., A Simplified Account of Polymorphic References, *Information Processing Letters*, 51, pp. 201–206, August 1994.

[SmVo95] Smith, G. and Volpano, D., Polymorphic Typing of Variables and References, to appear in *ACM Trans on Programming Languages and Systems*, 1996.

[Tof90] Tofte, M., Type Inference for Polymorphic References, *Information and Computation*, 89, pp. 1–34, 1990.